MACHINE LEARNING

MEI/1

University of Beira Interior,
Department of Informatics

Hugo Pedro Proença,
hugomcp@di.ubi.pt, 2024/25

Machine Learning

[06]

Syllabus

- Non-Linear Discrimination

- Multi-layer Perceptrons

- Convolutional Neural Networks

Linear Discriminants: Exercise

- Consider the following truth tables, corresponding to the classical "**AND**", ''**OR**'' and "**XOR**" problems:
  - Suppose we want to *learn* three **logistic regression** classifiers that <mark>appropriatelly discriminate between</mark> the "0"|"1" <mark>classes</mark>

AND

X1	X2	Y
0	0	0
0	1	0
1	0	0
1	1	1

✓

OR

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	1

✓

XOR

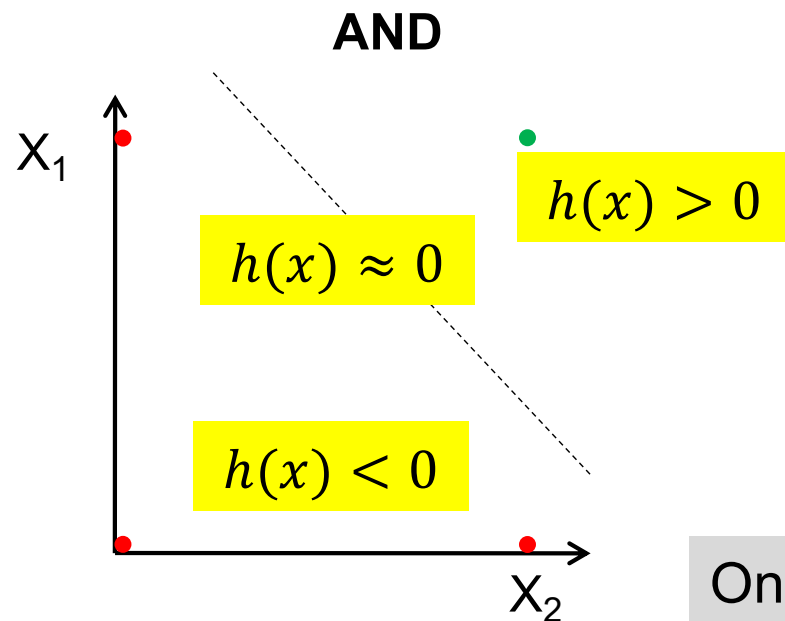| X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

✗

Linear Discriminants: Exercise

- As we previously saw, the logistic regression is only able **to find hyperplanes (straight lines, in 2D data)** that separate the subspaces of each class, which happens in the "AND/OR" problems.

  - These are called **linear discriminants**

AND

$h(x) > 0$

$h(x) \approx 0$

$h(x) < 0$

$$g\left(h_\theta(\boldsymbol{x})\right) = g(\theta_1 x_1 + \theta_2 x_2 + \theta_0)$$
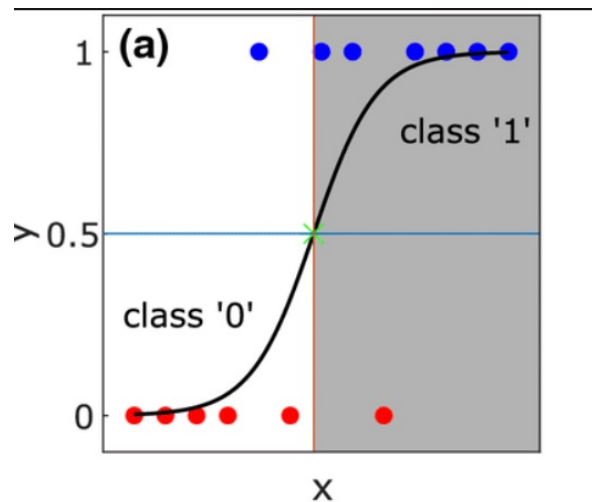
$$\frac{1}{1 + e^{-x}}$$

$$= \frac{1}{1 + e^{-(\theta_1 x_1 + \theta_2 x_2 + \theta_0)}}$$

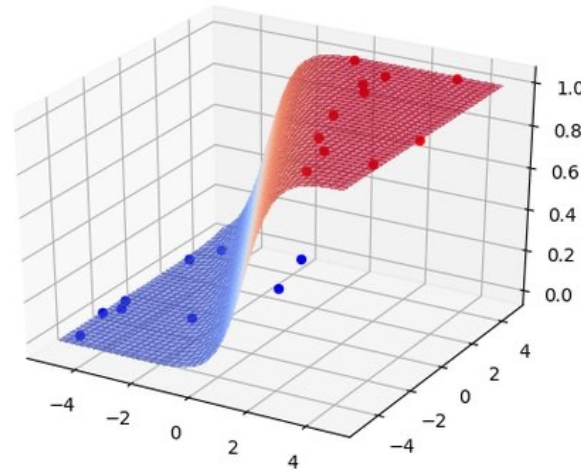One appropriate "AND" solution **could be**: $(\theta_1, \theta_2, \theta_0)$ = (0.8, 0.8, -1.5)

Linear Discriminants: Summary

- In short, one logistic regression model is effective only in linearly separable problems, where there **is a hyperplane** that **appropriately separates** the feature space.
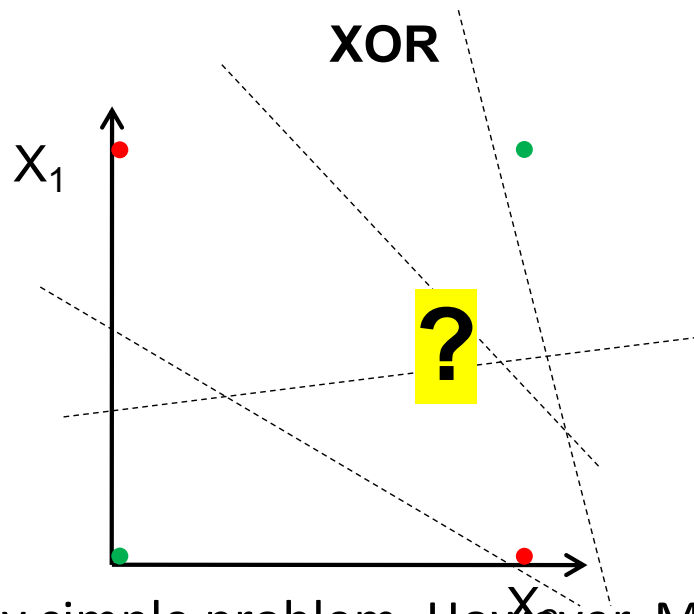

1D

2D

...nD

Linear Discriminants: Exercise

- However, for the "**XOR**" problem, there is no possible configurations for $\boldsymbol{\theta}$ that satisfy the requirements:

XOR

X_1

?

X_2

- **XOR** appears to be a very simple problem. However, Minksy and Papert (1969) showed that this was a big problem for neural network architectures of the 1960s, known as perceptrons.
  - The inefficiency of Perceptron networks to solve this problem caused the "*NN winter*" (period up to the early 90s, when NN were almost abandoned by the ML community)

Neural Networks

- Among the three classical approaches for machine learning (pattern recognition) models, this kind of methods aims at replicate the way the human brain works:
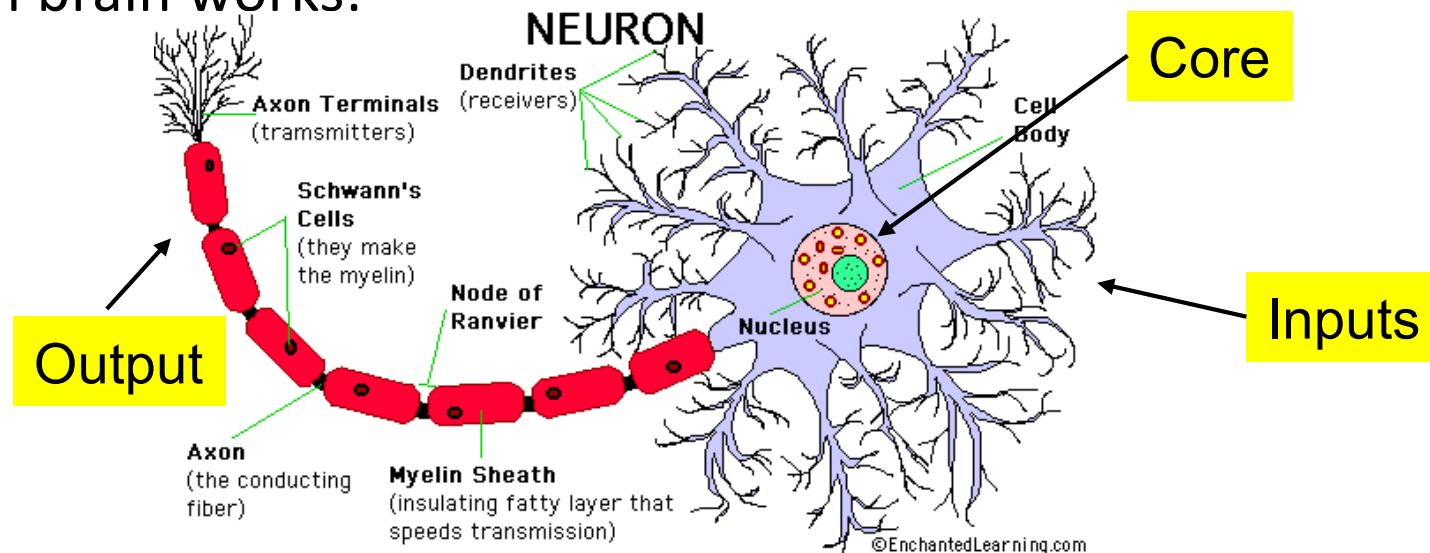
- In practice terms, this functioning model has remarkable similarities to the way our previous models were defined:
  - "*Mixing*" the values from a set of inputs, followed by one non-linear activation function".

Neural Networks
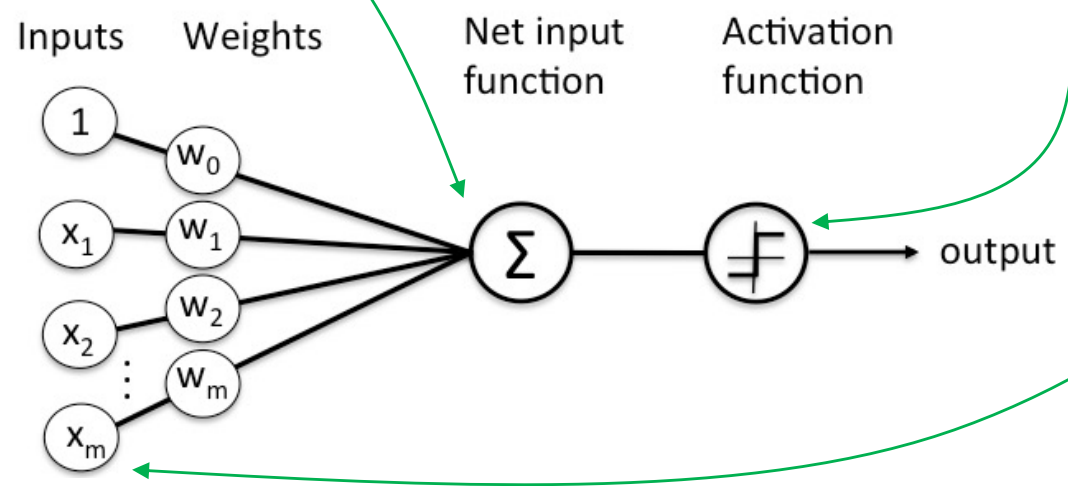
- A logistic regression classifier is defined by:

Inputs: x_1, x_2,...

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \frac{1}{1 + e^{-(\theta_1 x_1 + \theta_2 x_2 + \theta_0)}}$$

Phase 1: Convolution between **x** and $\boldsymbol{\theta}$

Phase 2: Non-linearity

- A Rosenblatt's perceptron is defined as:

Inputs	Weights	Net input function	Activation function

Σ

output

Neural Networks: MLP Architecture

- The key concept of the most classical kind of neural networks (**feed-forward**) is to define **multiple layers**, in which neurons of one layer receive the input of all neurons in the previous layer.
  - These are called neurons in hidden layers
  - Neurons in the first layer receive the **x** input
    - They are called neurons in the input layer
  - Neurons in the last layer provide the result of the model
    - They are called neurons in the output layer

inputs

classes

inputs /2 → # inputs * 2
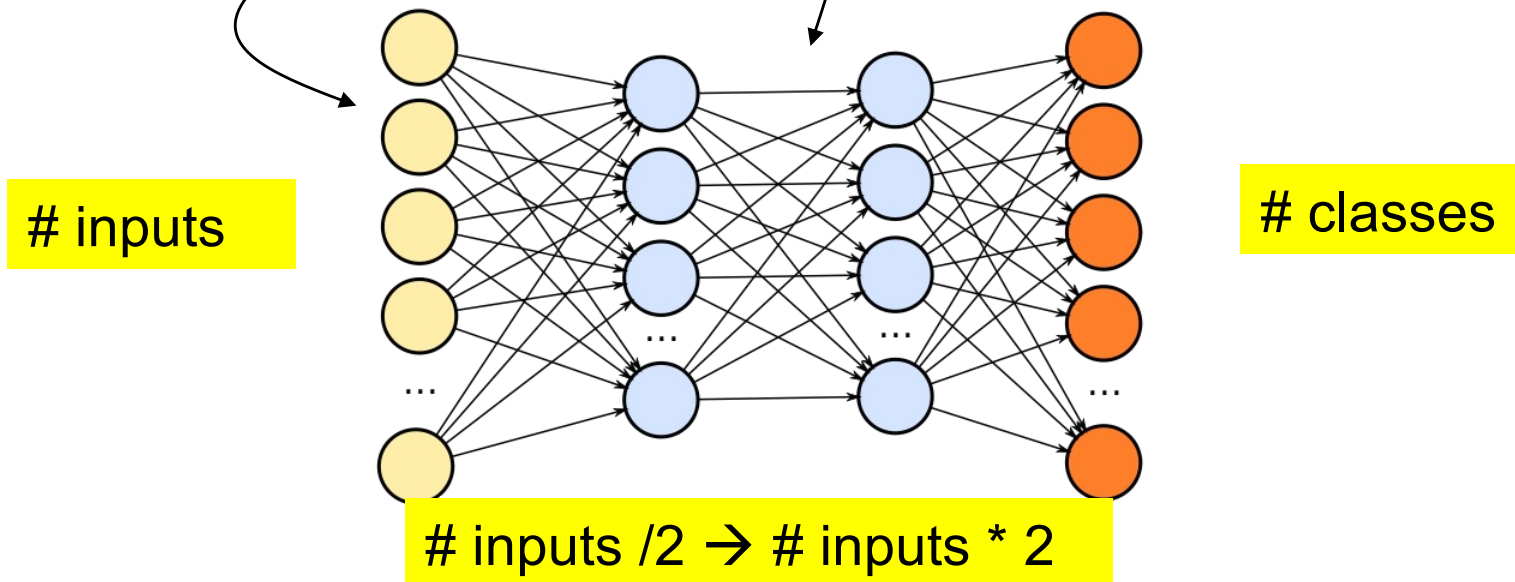
Machine Learning: Python **MLP**

- Let's start by the easiest part: (implementation)
  - How can I create one "Multi-Layer Perceptron" (MLP) network in Python and apply it to my problem?
  - **Step 1**: Import the corresponding library:

```
from sklearn.neural_network import MLPClassifier
```

  - **Step 2**: Have a **X** data set with shape (n, 2) and **y** with shape (n,)
    - In practice, **X** will be a "list of lists" and **y** will be a list.

```
X = [[0., 0.], [1., 1.]]
y = [0, 1]
```

  - **Step 3**: Create the network:

```
clf = MLPClassifier(solver='lbfgs', alpha=1e-5,
                    hidden_layer_sizes=(5, 2), random_state=1)
```

  - **Step 4**: Start learning:

```
clf.fit(X, y)
```

  - **Step 5**: Use it, to predict on new instances:

```
clf.predict([[2., 2.], [-1., -2.]])
```

# Neural Networks

- When designing a neural network, there are different parameterizations that have to be chosen, with might determine the system effectiveness:
  - The ==number of neurons== in the input/output layers result directly of the problem considered:
    - **Input Layer = Dimension of the Feature Space**
    - **Output Layer = Number of classes** (==hot encoded==)

  1      0 0 1
  2 ⟹ 0 1 0
  3      1 0 0
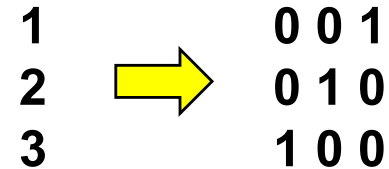
- In the hidden layers, the number of neurons can vary:
  - A ==too short== number might not be enough to model the decision surface desired;
  - A ==too high== value might lead to ==overfitting==
  - In practice, values between half and the double of the number of neurons in the input layer are tested

- Regarding the number of hidden layers:
  Networks with ==one layer== have the ability to approximate any linear decision surface
  Networks with ==two layers== approximate any continuous decision surface
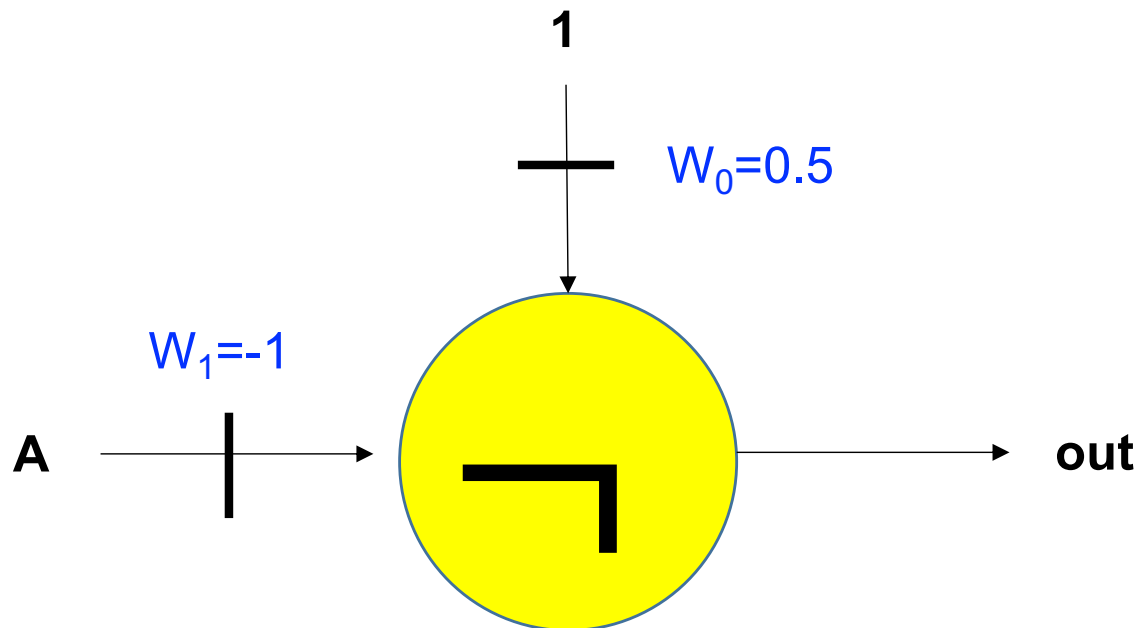  Networks with ==three layers== approximate any decision surface
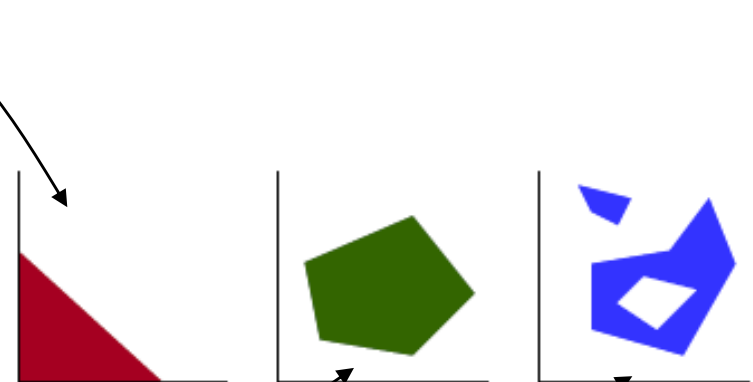
# Machine Learning: NN Exercise

- Considering that:

$$A \bigotimes B = \neg \left( (A \bigwedge B) \vee (\neg A \bigwedge \neg B) \right)$$

- For example, how to infer the weights for a "NOT" neuron, i.e., a neuron that replicates the functioning of a logical "NOT" operation.
    - In this simple case, there are various weight configurations that will work
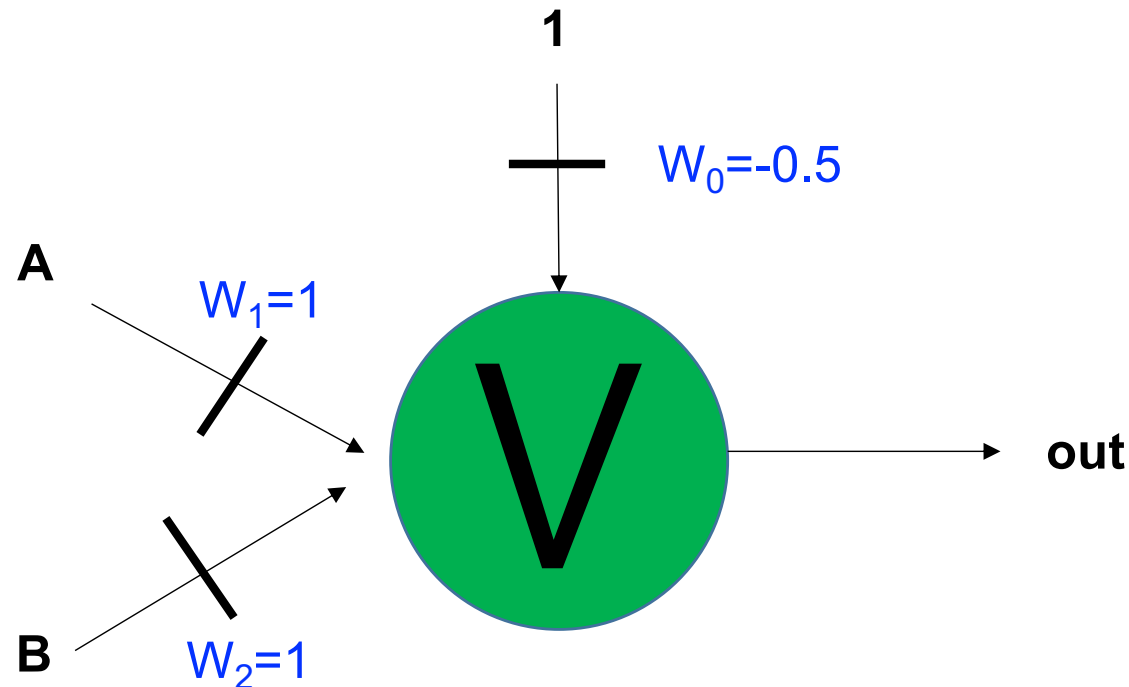
# Machine Learning: NN Example

- Considering that:

$$A \bigotimes B = \neg \left( (A \bigwedge B) \vee (\neg A \bigwedge \neg B) \right)$$

- Now, how to infer the weights for a "OR" neuron, i.e., a neuron that replicates the functioning of a logical "OR" operation.
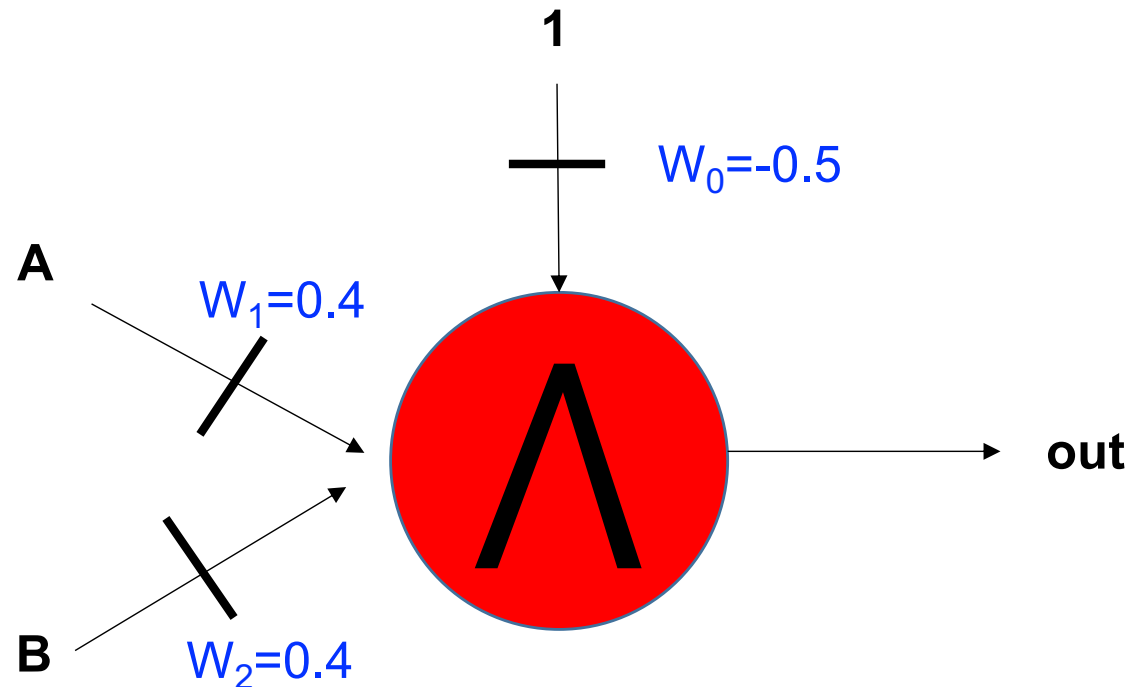  - Again, there are various weight configurations that will work:

**1**

$W_0 = -0.5$

**A**

$W_1 = 1$

**B**

$W_2 = 1$

**out**

# Machine Learning: NN Example

- Considering that:

$$A \bigotimes B = \neg ( (A \bigwedge B) \vee (\neg A \bigwedge \neg B) )$$

- Next, in a similar way, if we want to infer the weights for a "AND" neuron, i.e., a neuron that replicates the functioning of a logical "AND" operation.
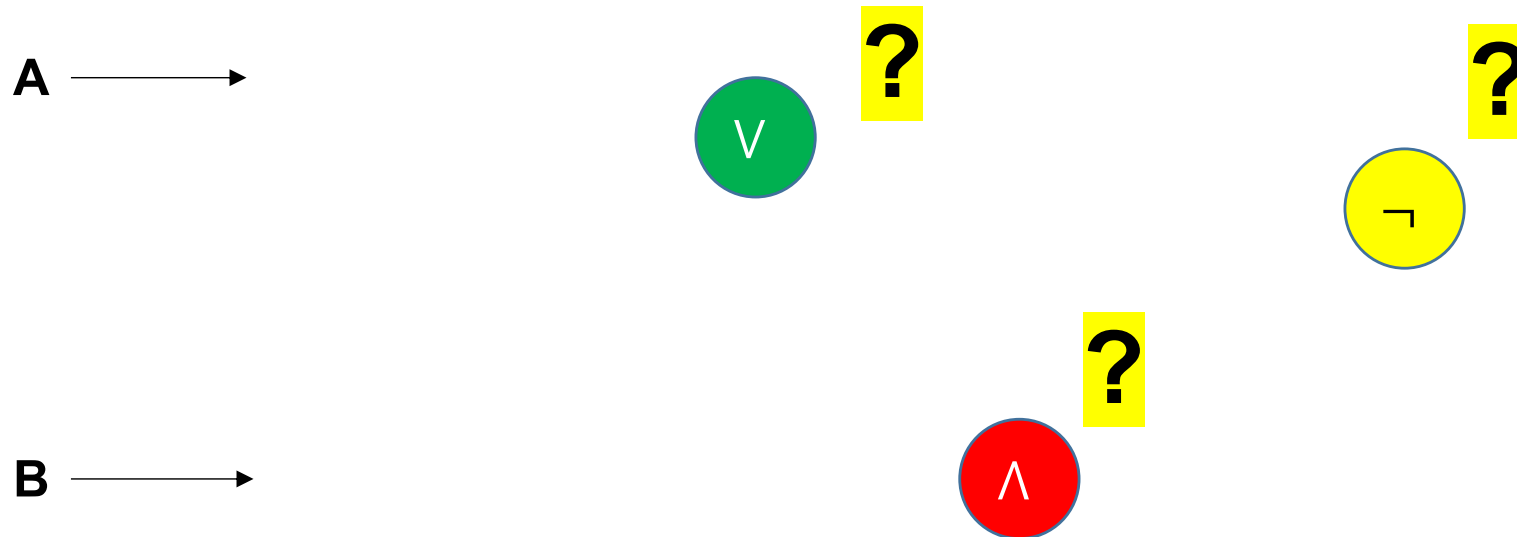    - As in the previous cases, there are various weight configurations that will work:

**1**

$W_0 = -0.5$

**A**

$W_1 = 0.4$

**B**

$W_2 = 0.4$

**out**

# Machine Learning: NN Exercise

- Considering that:

$$A \bigotimes B = \neg \left( (A \bigwedge B) \vee (\neg A \bigwedge \neg B) \right)$$

- Design a multi-layer network, with the corresponding weights $\boldsymbol{\theta}$, able to solve the "**XOR**" problem.
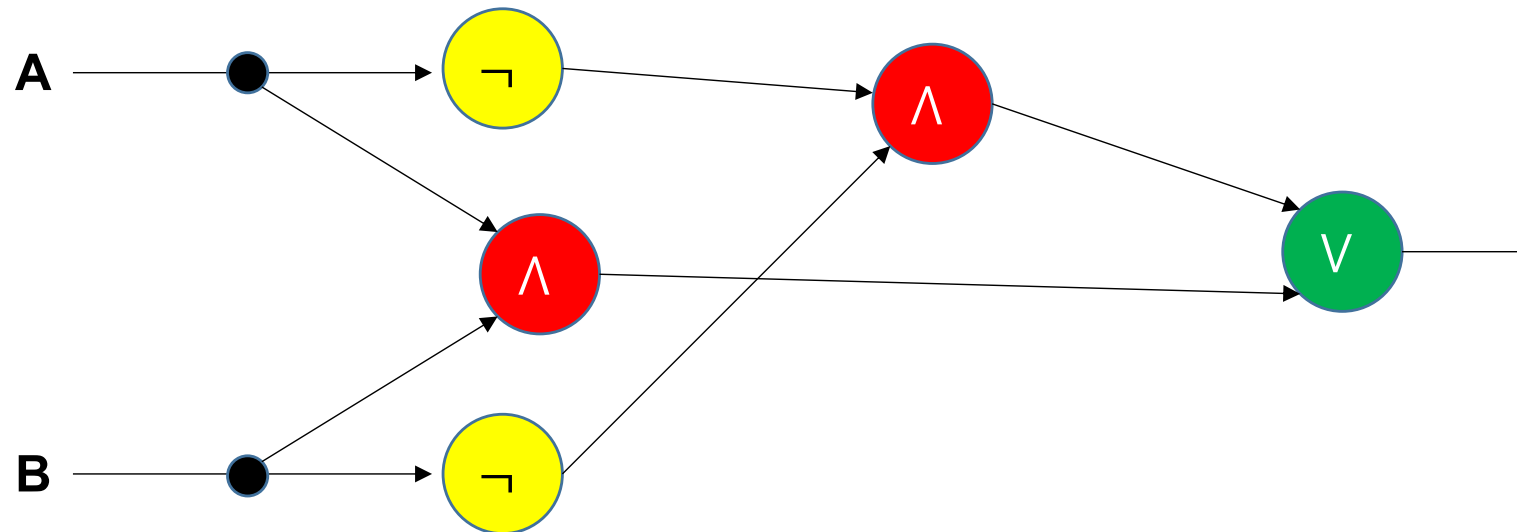
# Machine Learning: NN Exercise

- Considering that:

$$A \bigotimes B = \neg \left( (A \bigwedge B) \vee (\neg A \bigwedge \neg B) \right)$$

- Design a multi-layer network, with the corresponding weights $\boldsymbol{\theta}$, able to solve the "**XOR**" problem.



- <u>This will be a network "specific" to reproduce this function.</u>

- <u>However, the big question remains: **How to automatically obtain the $\boldsymbol{\theta}$ values?**</u>

# Neural Networks: Learning

- In case of multilayered networks, the closed-form equation for the whole network, the cost function and the corresponding derivatives might not be easy to obtain.

- **Exercise**:
  - Obtain the function that describes the functioning of the following network, considering that the transfer functions of all nodes.
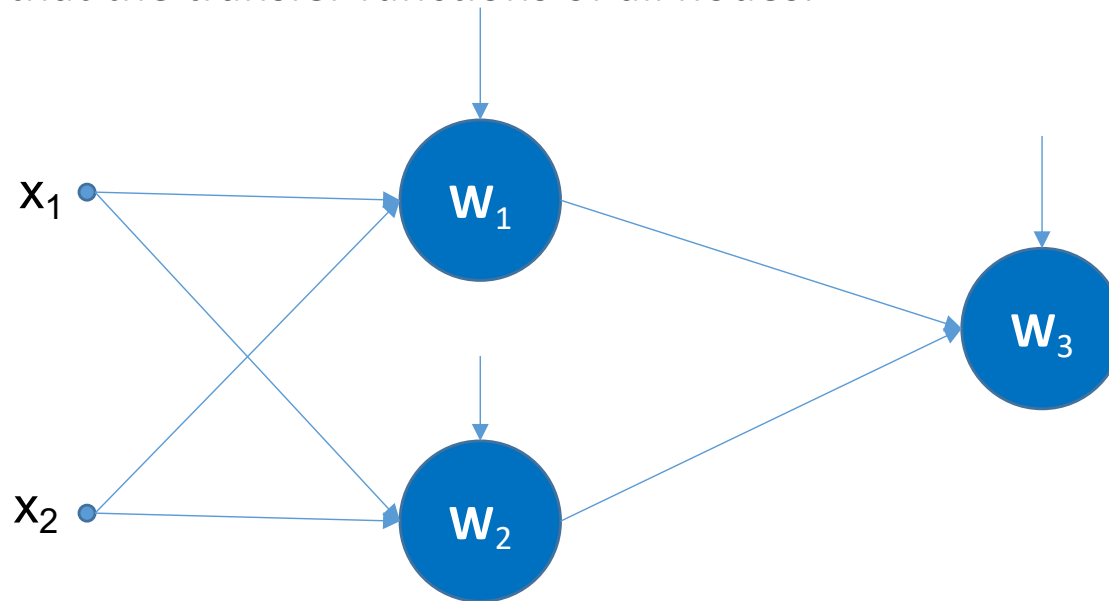
# Backpropagation
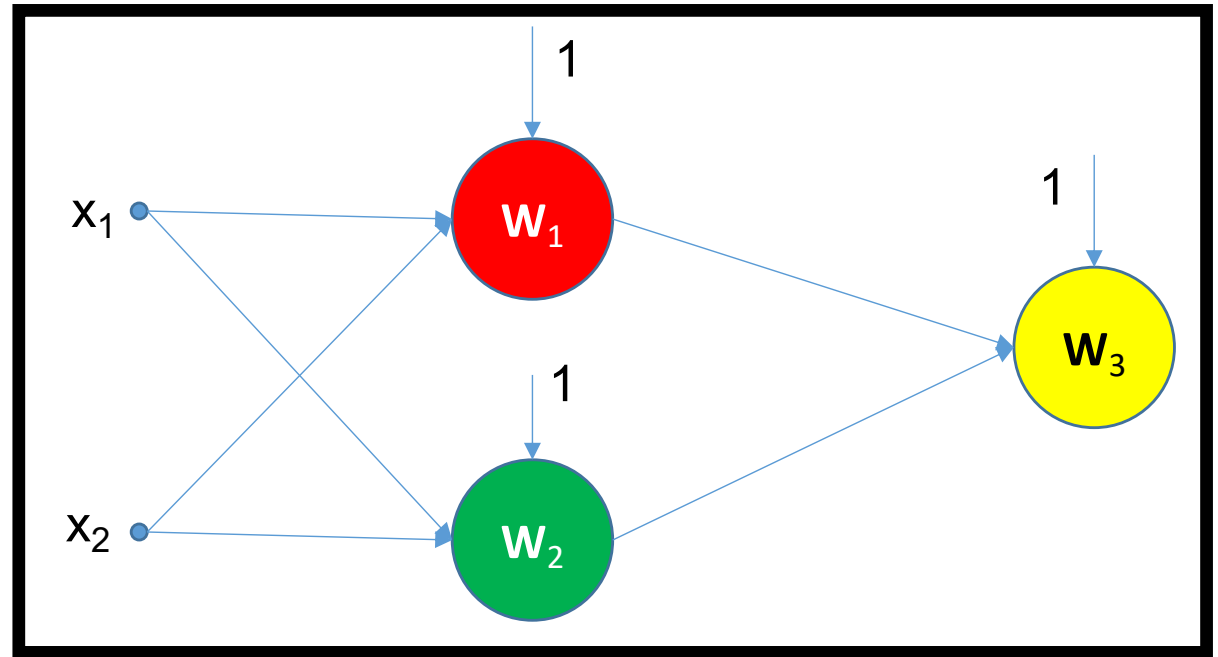


$$t_1 = \frac{1}{1 + e^{-w_{1,0} - w_{1,1}x_1 - w_{1,2}x_2}}$$

$$t_2 = \frac{1}{1 + e^{-w_{2,0} - w_{2,1}x_1 - w_{2,2}x_2}}$$

$$\text{NN} = \frac{1}{1 + e^{-w_{3,0} - w_{3,1}t_1 - w_{2,2}t_2}}$$

$$\text{NN} = \frac{1}{1 + e^{-w_{3,0} - w_{3,1}\frac{1}{1+e^{-w_{1,0}-w_{1,1}x_1-w_{1,2}x_2}} - w_{3,2}\frac{1}{1+e^{-w_{2,0}-w_{2,1}x_1-w_{2,2}x_2}}}}$$

# Backpropagation

$$NN = \frac{1}{1 + e^{-w_{3,0} - w_{3,1} \frac{1}{1+e^{-w_{1,0} - w_{1,1} x_1 - w_{1,2} x_2}} - w_{3,2} \frac{1}{1+e^{-w_{2,0} - w_{2,1} x_1 - w_{2,2} x_2}}}}$$

$$J(\mathbf{w}) = \frac{1}{N} \sum Cost(NN(\omega, x^{(i)}, y^{(i)}))$$

- $Cost\left(NN(\mathbf{w}, x^{(i)}), y^{(i)}\right) =$

  $$\begin{cases} -\log\left(NN(\mathbf{w}, x^{(i)})\right), \text{ if } y^{(i)}=1 \\ \\ -\log\left(1 - NN(\mathbf{w}, x^{(i)})\right), \text{ if } y^{(i)}=0 \end{cases}$$

- Therefore, as we did before for the logistic regression classifier, the cost function is combined in a single function:

  $$J(\mathbf{w}) = -\frac{1}{N} \sum_i y^{(i)} \log\left(NN(\mathbf{w}, x^{(i))}\right) + (1-y^{(i))}) \log\left(1 - NN(\mathbf{w}, x^{(i))})\right)$$

# Backpropagation

- Using the gradient descent (delta rule) learning strategy previously described, it will be required to obtain:

$$\frac{\partial}{\partial w_{1,0}} = ? \qquad\qquad \frac{\partial}{\partial w_{2,0}} = ? \qquad\qquad \frac{\partial}{\partial w_{3,0}} = ?$$

$$\frac{\partial}{\partial w_{1,1}} = ? \qquad\qquad \frac{\partial}{\partial w_{2,1}} = ? \qquad\qquad \frac{\partial}{\partial w_{3,1}} = ?$$

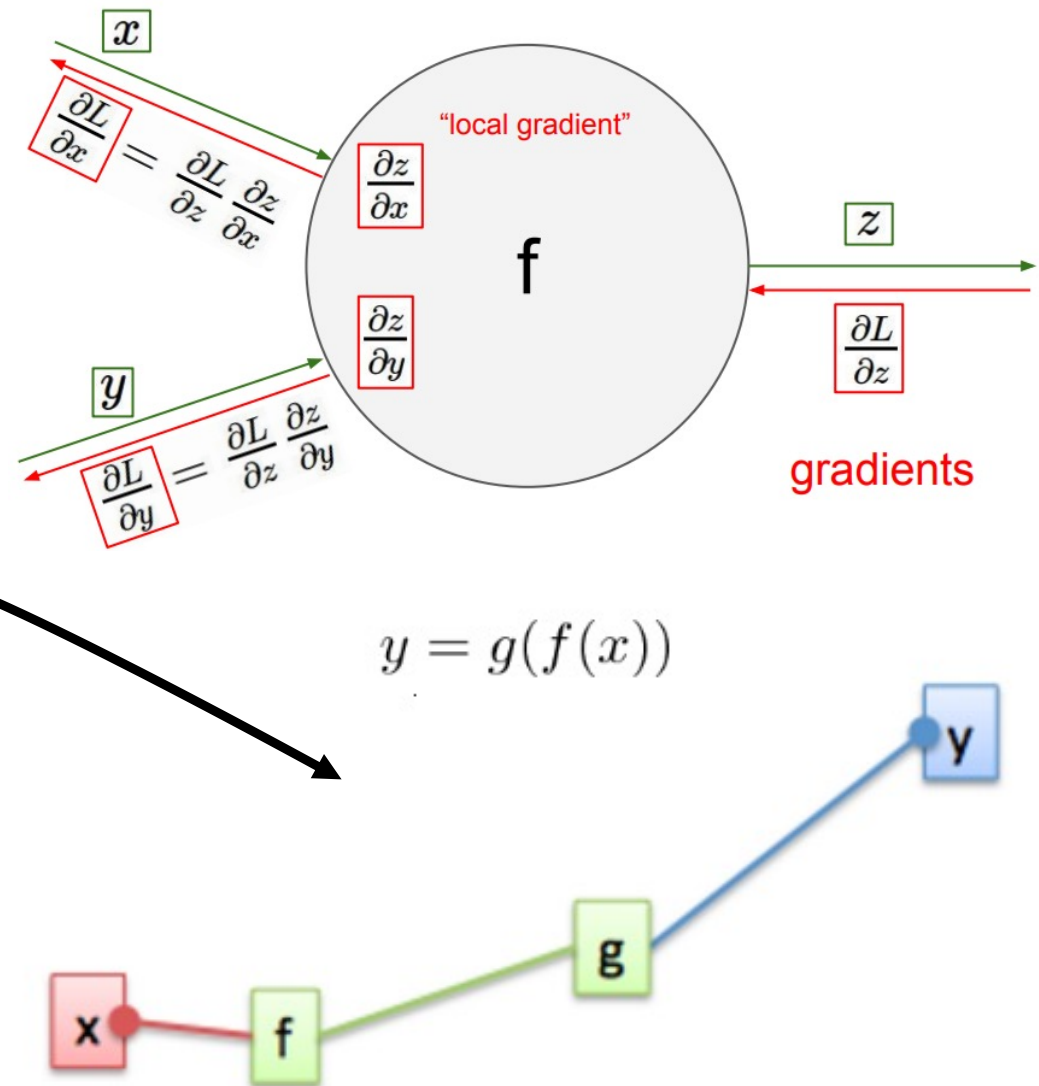$$\frac{\partial}{\partial w_{1,2}} = ? \qquad\qquad \frac{\partial}{\partial w_{2,2}} = ? \qquad\qquad \frac{\partial}{\partial w_{3,2}} = ?$$

...and this is a tiny network...

# Backpropagation and the Chain Rule

- **"Backpropagation"** is the short name for "backward propagation of errors"

- It is an algorithm for supervised learning of multi-layer artificial neural networks, based in gradient descent

- The key concept is the **chain rule**:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial g} \cdot \frac{\partial g}{\partial f} \cdot \frac{\partial f}{\partial x}$$

- Calculates the gradient of the error function with respect to the neural network's weights;

- It is a **generalization** of the delta rule for perceptrons to multilayer feed-forward neural networks.

$x$

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$$

"local gradient"

$\frac{\partial z}{\partial x}$

$f$

$\frac{\partial z}{\partial y}$

$z$

$\frac{\partial L}{\partial z}$

$y$

$$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$$

gradients

$$y = g(f(x))$$

x  f  g  y

# Convolutional Neural Networks (CNNs)

- CNNs are a type of Neural Networks that have been augmenting their popularity in most tasks related to Computer Vision
  - E.g., Image Segmentation, Classification.
- The property of **shift invariance** gives them the **biological inspiration** of the human visual system and keeps the number of weights relatively small, making learning a feasible task.
- In opposition to traditional Feed-forward nets, neurons in CNNs are arranged in **three dimensions**.
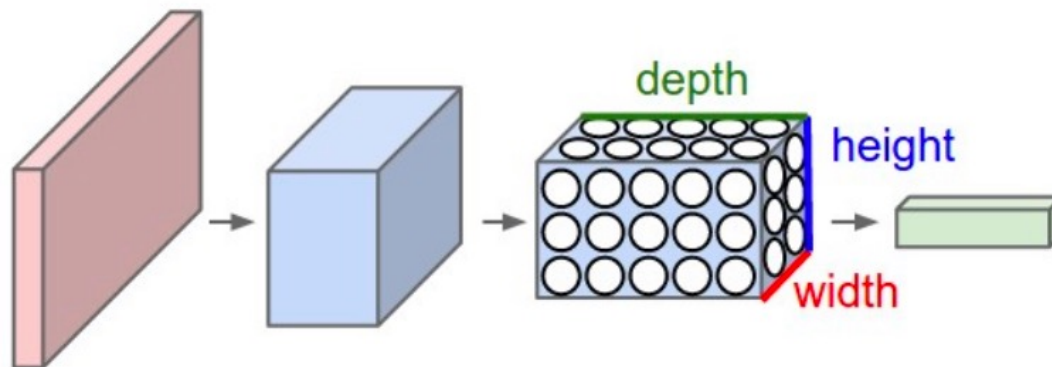
# Convolutional Neural Networks (CNNs)

- Each layer of a CNN transforms a 3D input into a 3D output.
- This pioneering work in CNNs was due to Yann LeCun (LeNet5) after many previous successful iterations since 1988.
- Initially, the **LeNet** architecture was used mainly for character recognition tasks such as reading zip codes, digits...
- The efficacy of CNNs in visual tasks is the main reason behind the popularity of deep learning. They are powering major advances in computer vision, with applications for robotics, security and medical diagnosis.
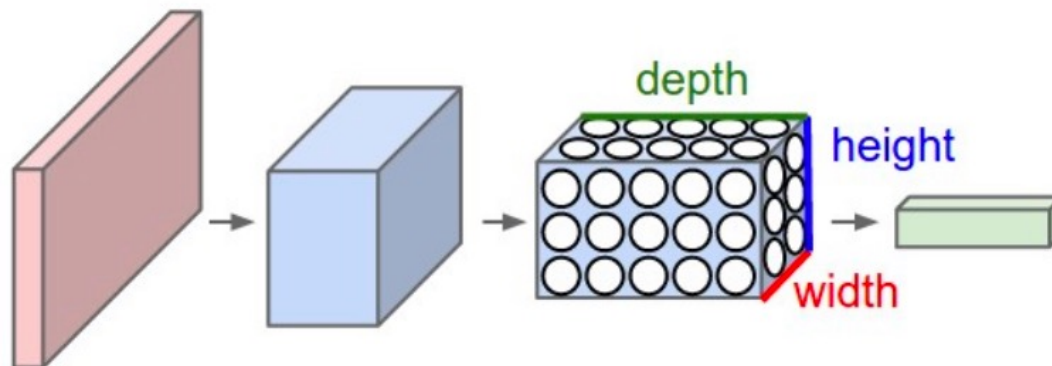
# Convolutional Neural Networks (CNNs)

- The most typical structure of a CNN is:
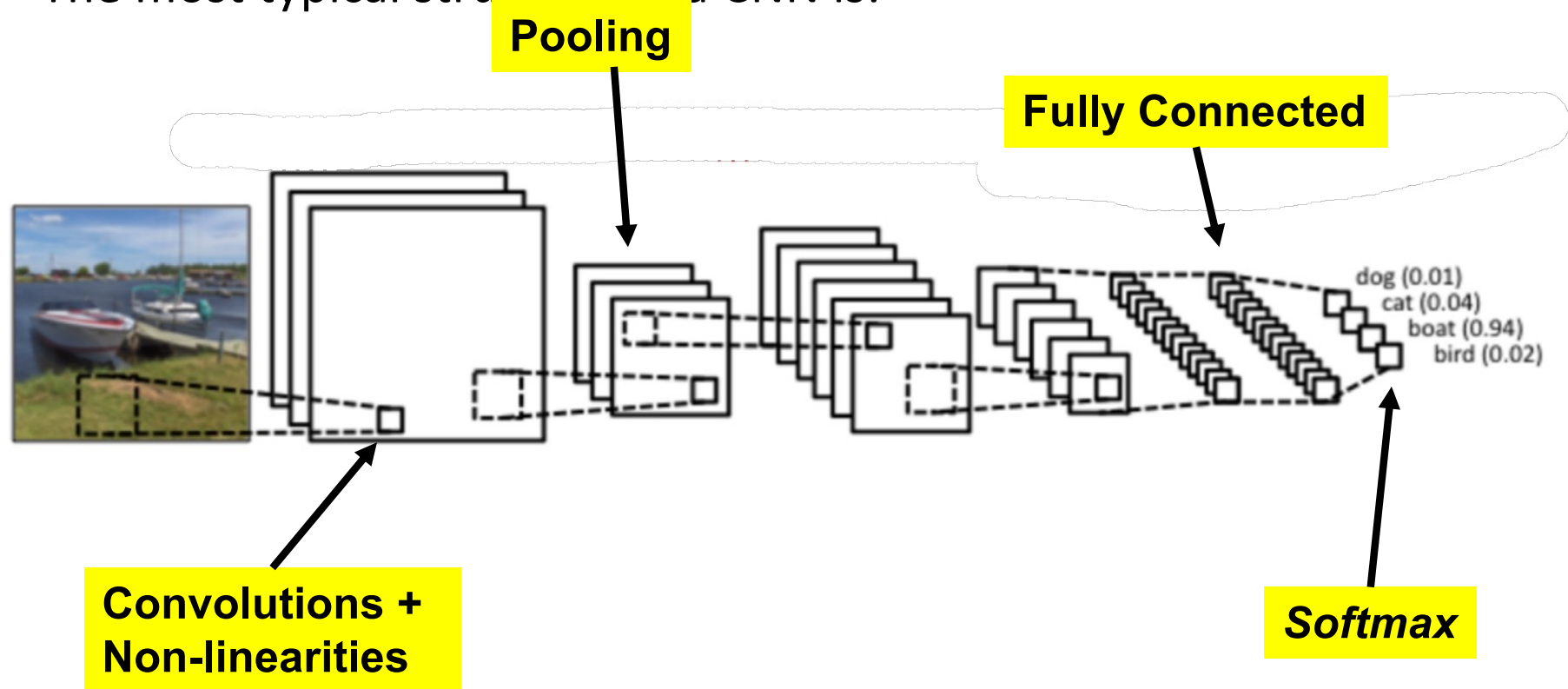


These operations are the basic building blocks of *most CNNs*, so understanding how these work is an important step to understand the functioning of these powerful models.

# Signals and Systems

❑ What is a **signal**?

  ❑ It can be regarded as a **description how** a **parameter varies** (dependent variable) **with respect to another** (independent variable);

  ❑ E.g., the voltage of an electric charge varies with respect to time (1D signals) ;

  ❑ E.g., the intensity of a pixel varies with respect its location in image (2D signals);

❑ Tipically, signals are denoted by **upper case letters**.

  ❑ Discrete signals are denoted by **[]**:

    ❑ E.g., X[n], Y[k]

  ❑ Continuous signals are denoted by ()

    ❑ E.g., X(i), Y(j)

# Convolution

❑It is a mathematical operation that describes the relationship between three signals:
  ❑One **input** signal;
  ❑One **impulse response;**
  ❑Yielding the **output** signal.

❑It can be seen as "***extracting a specific feature from a signal, depending on the filter used***".

❑As it combines addition (**+**) with multiplication (**x**), it is usually denoted by "***\****".

  ❑Y[k]=H[k]*X[k]
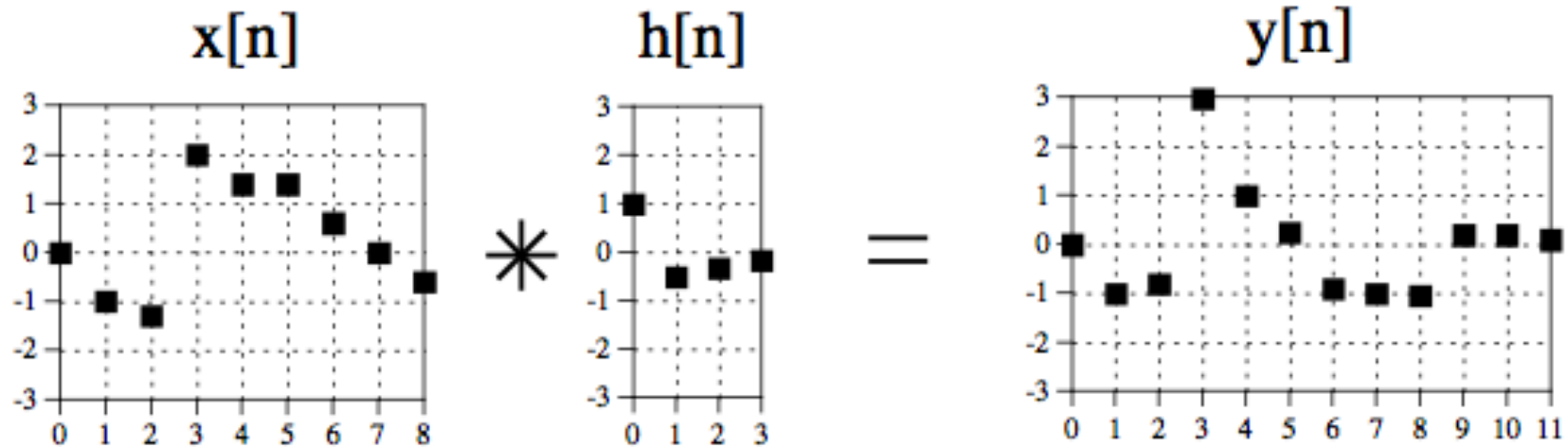
$$y[i] = \sum_{j=0}^{M-1} h[j] x[i-j]$$

# Convolution: Exercise

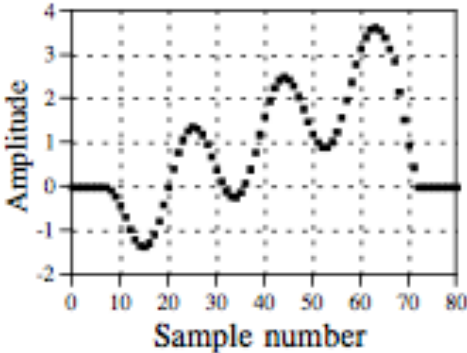☐Obtain the result of the convolution of the following signals:
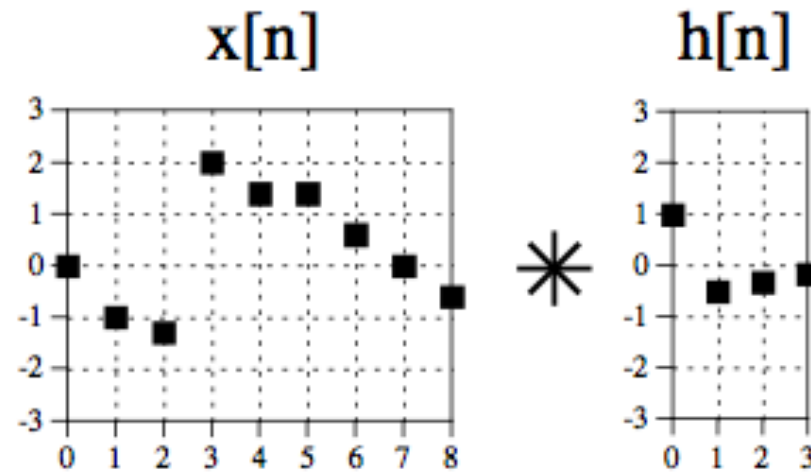
# Convolution: Exercise (cont.)

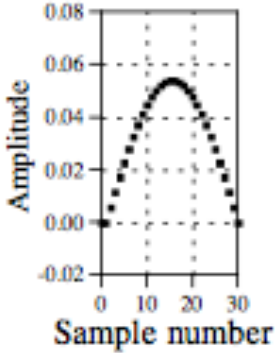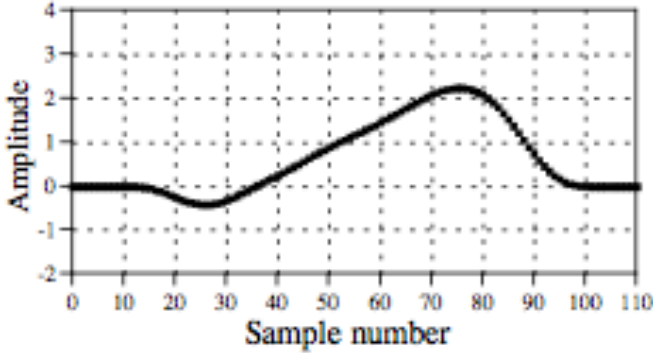❑Obtain the result of the convolution of the following signals:

# Convolution: Examples

**❑Low-pass filtering:**

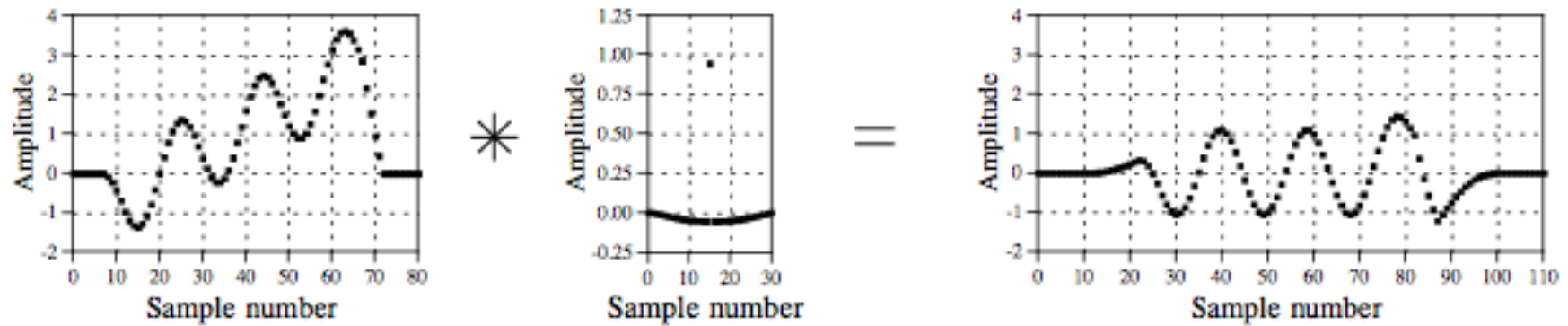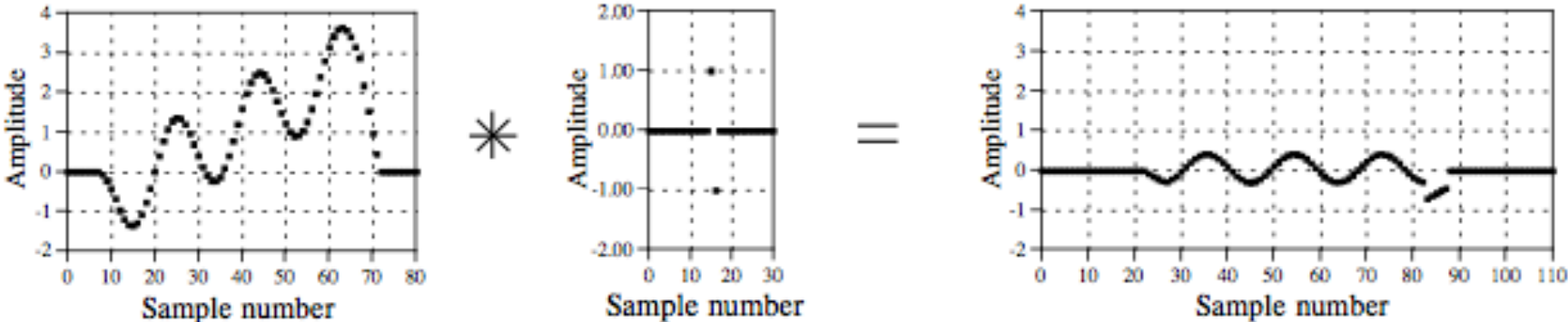# Convolution: Examples

❑**High-pass filtering:**

# Convolution: Examples

❑**Discrete derivative:**

# Convolution: Caution!!

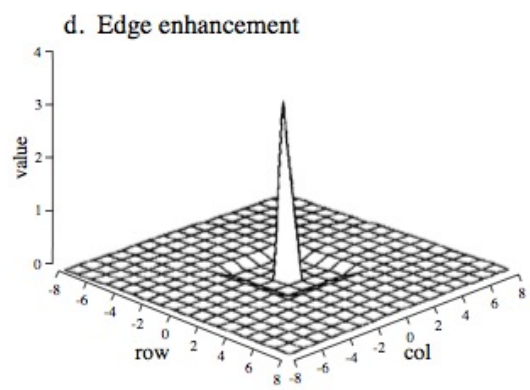❑When an input signal is convolved with an impulse response of length "M", then the first and last "M-1" components are not fully reliable.

❑Why is this?

# Filters: Examples



a. Pillbox

b. Gaussian

c. Square

d. Edge enhancement

e. Sinc
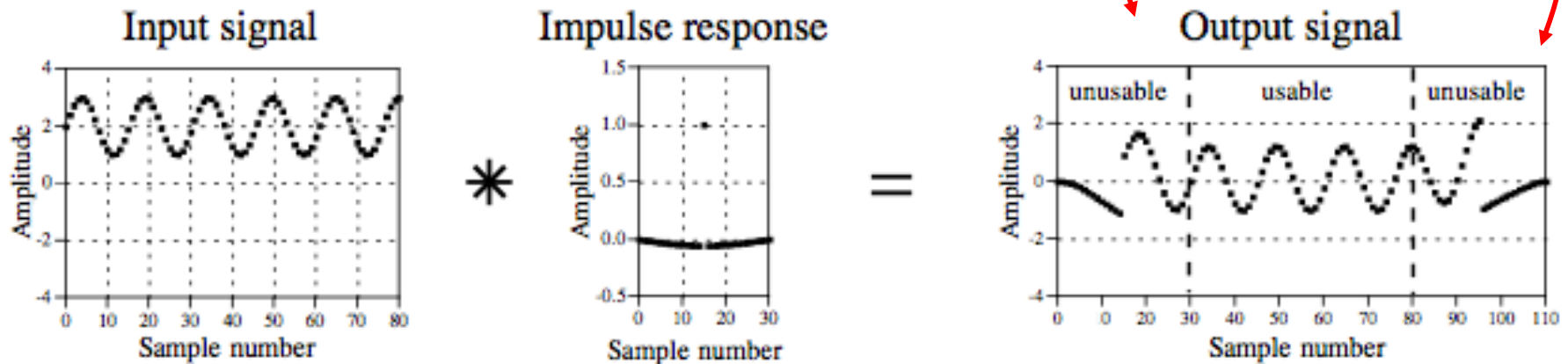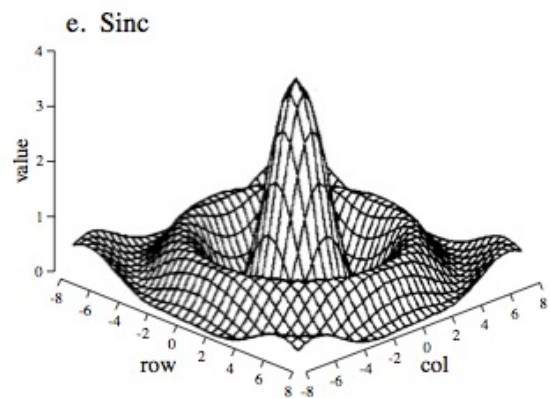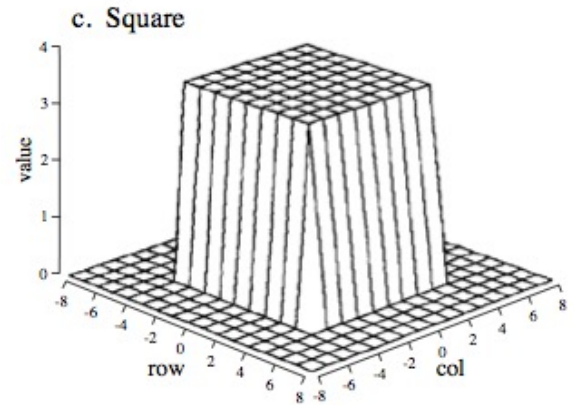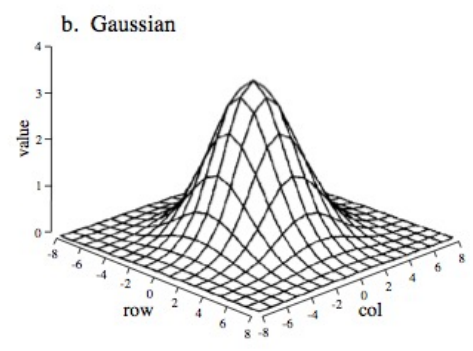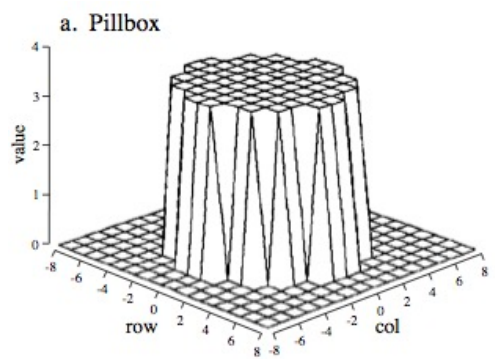
# Convolutional Neural Networks (CNNs)

- **Convolution**
  - This block computes the convolution between an input map **x** with a bank of k multi-dimensional filters **f,** to obtain the results **y.**

$$\mathbf{x} \in \mathbb{R}^{H \times W \times D}, \quad \mathbf{f} \in \mathbb{R}^{H' \times W' \times D \times D''}, \quad \mathbf{y} \in \mathbb{R}^{H'' \times W'' \times D''}.$$

- Formally, the outputs **y** are given by:

$$y_{i''j''d''} = b_{d''} + \sum_{i'=1}^{H'} \sum_{j'=1}^{W'} \sum_{d'=1}^{D} f_{i'j'd} \times x_{i''+i'-1,j''+j'-1,d',d''}.$$

# Convolutional Neural Networks (CNNs)

- Convolution (padding and stride)
  - Usually it is possible to specify top, bottom, left, right paddings ($P_h^-$, $P_h^+$, $P_w^-$, $P_w^+$) of the input array and subsampling strides ($S_h$, $S_w$) of the output array.

$$y_{i''j''d''} = b_{d''} + \sum_{i'=1}^{H'}\sum_{j'=1}^{W'}\sum_{d'=1}^{D} f_{i'j'd} \times x_{S_h(i''-1)+i'-P_h^-,\, S_w(j''-1)+j'-P_w^-,\, d',\, d''}.$$
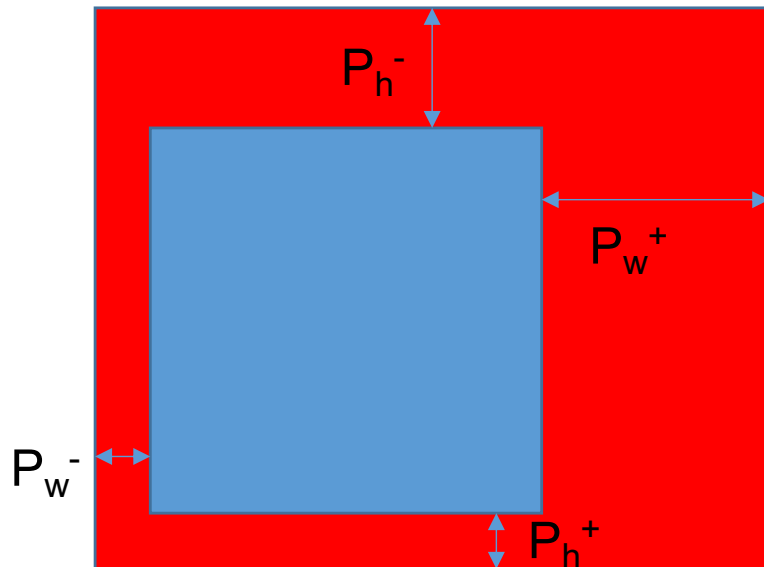
The output size is given by:

$$H'' = 1 + \left\lfloor \frac{H - H' + P_h^- + P_h^+}{S_h} \right\rfloor.$$

# Convolutional Neural Networks (CNNs)

- **Spatial Pooling**
  - The typical blocks are the max and sum pooling, respectively computing the maximum and the summed response of each feature channel in a H' x W' patch.
- Pooling progressively reduces the spatial size of the input representation.
  - This reduces the number of parameters and, therefore, controls over fitting;
  - Also, it makes the network invariant to small transforms, distortions and translations in the input image (a small distortion in input will not change the output of pooling).

$$y_{i''j''d} = \max_{1 \le i' \le H', 1 \le j' \le W'} x_{i''+i'-1, j''+j'-1, d}. \qquad y_{i''j''d} = \frac{1}{W'H'} \sum_{1 \le i' \le H', 1 \le j' \le W'} x_{i''+i'-1, j''+j'-1, d}.$$

# Convolutional Neural Networks (CNNs)
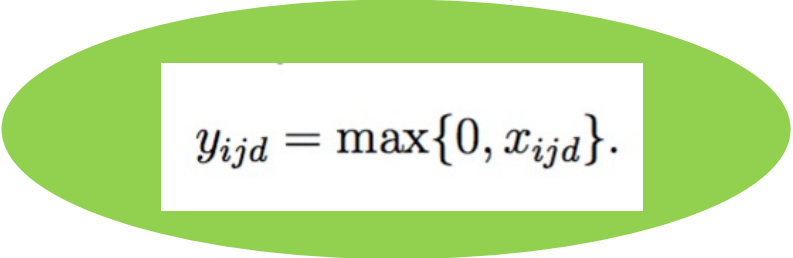
- **Pooling**
  - Note that Pooling down samples the input volume only spatially;
  - The input depth is equal to the output depth;
  - The pooling operation is often considered **deprecated**. To reduce the size of the representation, in is possible to use larger strides in the convolution layers.



Example: max() pooling
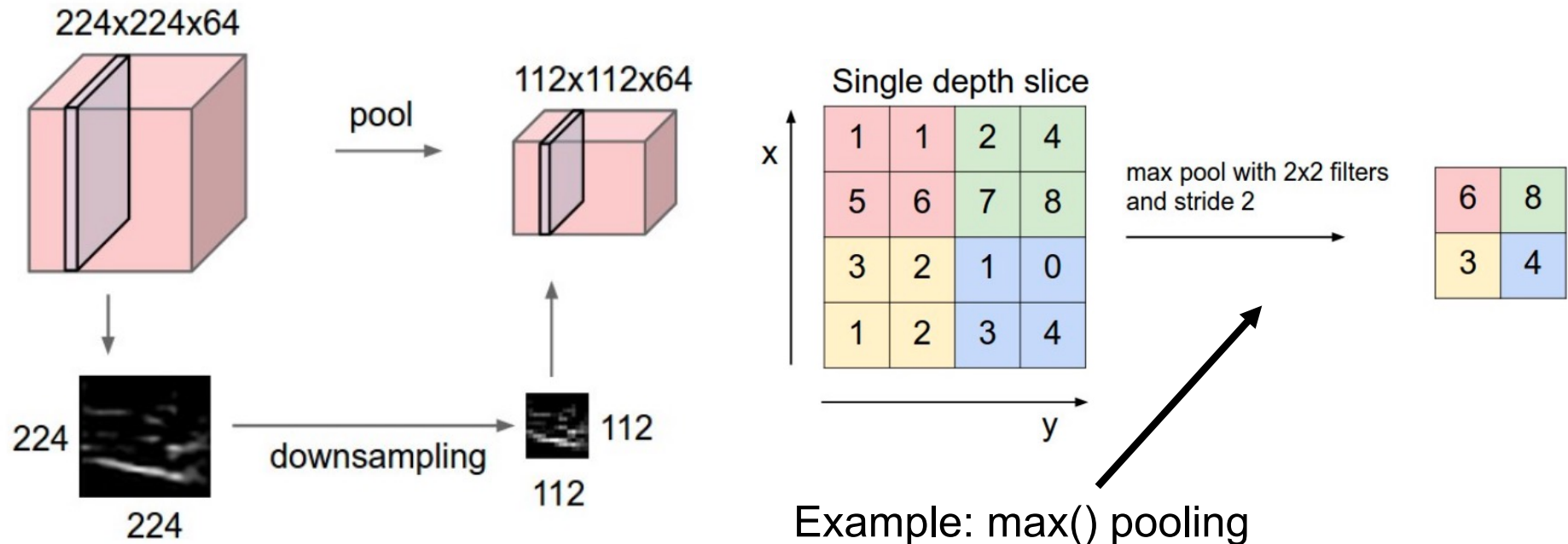
# Convolutional Neural Networks (CNNs)

- **Non-Linearity**
  - There are two basic non-linear activation functions used in CNNS: "ReLU" (Rectified Linear Units) and "Sigmoid".

$$y_{ijd} = \max\{0, x_{ijd}\}.$$

$$y_{ijd} = \sigma(x_{ijd}) = \frac{1}{1 + e^{-x_{ijd}}}.$$

  - As advantages with respect to each other, Sigmoid is consider not to blow up activation, while ReLU **does not vanishes the gradient**
    - In the case of Sigmoid, when the input grows to infinitely large, the derivative tends to 0.
  - However, in the case of ReLU, there is no mechanism to constrain the output of the neuron, as the input is often the output)

# Convolutional Neural Networks (CNNs)

- **Fully Connected layers**
  - Neurons in a fully connected layer have full connections to all activations in the previous layer, as in a regular feed-forward network.
  - In practical terms, these neurons resemble pretty much the neurons in "Convolution" layers.
    - The only difference between fully connected and Convolution layers is that the neurons in the former layer are connected only to a local region in the input, and that many of the neurons in a CONV volume share parameters.
    - However, the neurons in both layers still compute dot products, so their functional form is identical.
  - For example, an FC layer with K=4096 that is looking at some input volume of size 7×7×512 can be expressed as a Convolution layer with F=7 x 7 x 4096 (padding 0, stride 1).
  - In other words, we are setting the filter size to be exactly the size of the input volume;
  - Hence the output will simply be 1×1×4096.

# Convolutional Neural Networks (CNNs)

- **Softmax**
  - Can be seen as the combination of an activation function (exponential) and a normalization operator.
  - It is usually applied as the transfer function of the last layer of the CNN, where the idea is to push up the maximum value of the responses to "1", and all the other values to "0".
  - In practice, it simulates the probability of the input corresponding to each category, represented by a neuron in the output layer.

$$y_{ijk} = \frac{e^{x_{ijk}}}{\sum_{t=1}^{D} e^{x_{ijt}}}.$$

# Convolutional Neural Networks (CNNs)

- Most of the data memory used by CNNs is used in the early Convolutional layers (where spatial resolution is maximal), whereas most of the parameters of the network are in the fully connected layers.
  - Example **VGGNet,** one of the well known and succeeded architectures:

INPUT: [224x224x3] memory: 224*224*3=150K weights: 0
CONV3-64: [224x224x64] memory: 224*224*64=3.2M weights: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M weights: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K weights: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M weights: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M weights: (3*3*128)*128 = 147,456 POOL2: [56x56x128]
memory: 56*56*128=400K weights: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K weights: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K weights: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K weights: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K weights: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K weights: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K weights: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K weights: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K weights: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K weights: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K weights: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K weights: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K weights: 0
FC: [1x1x4096] memory: 4096 weights: 7*7*512*4096 = 102,760,448
FC: [1x1x4096] memory: 4096 weights: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 weights: 4096*1000 = 4,096,000

# Convolutional Neural Networks (CNNs)

- **VGGNet:**
  - The total memory used is about 4 bytes * 24,000,000 = 93 MB
  - This is required only for the **forward step**
  - In practice, the backward step requires around the double memory;
  - The network has 138,000,000 parameters to be tuned by the back-propagation algorithm.
- It should be noted that the conventional paradigm of a linear list of layers is not the state-of-the-art anymore.
  - Google's Inception architectures and also Residual Networks from Microsoft Research Asia.
  - Both of these feature more intricate and different connectivity structures.
- Most of the COTS (commercial off-the-shelf) models have complex graph-based architectures.

# Convolutional Neural Networks (CNNs)
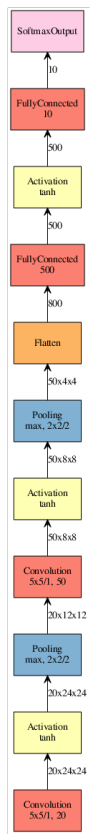
- Accuracy vs. Number of operations for a single forward step. Circumference radii corresponds to the number of parameters
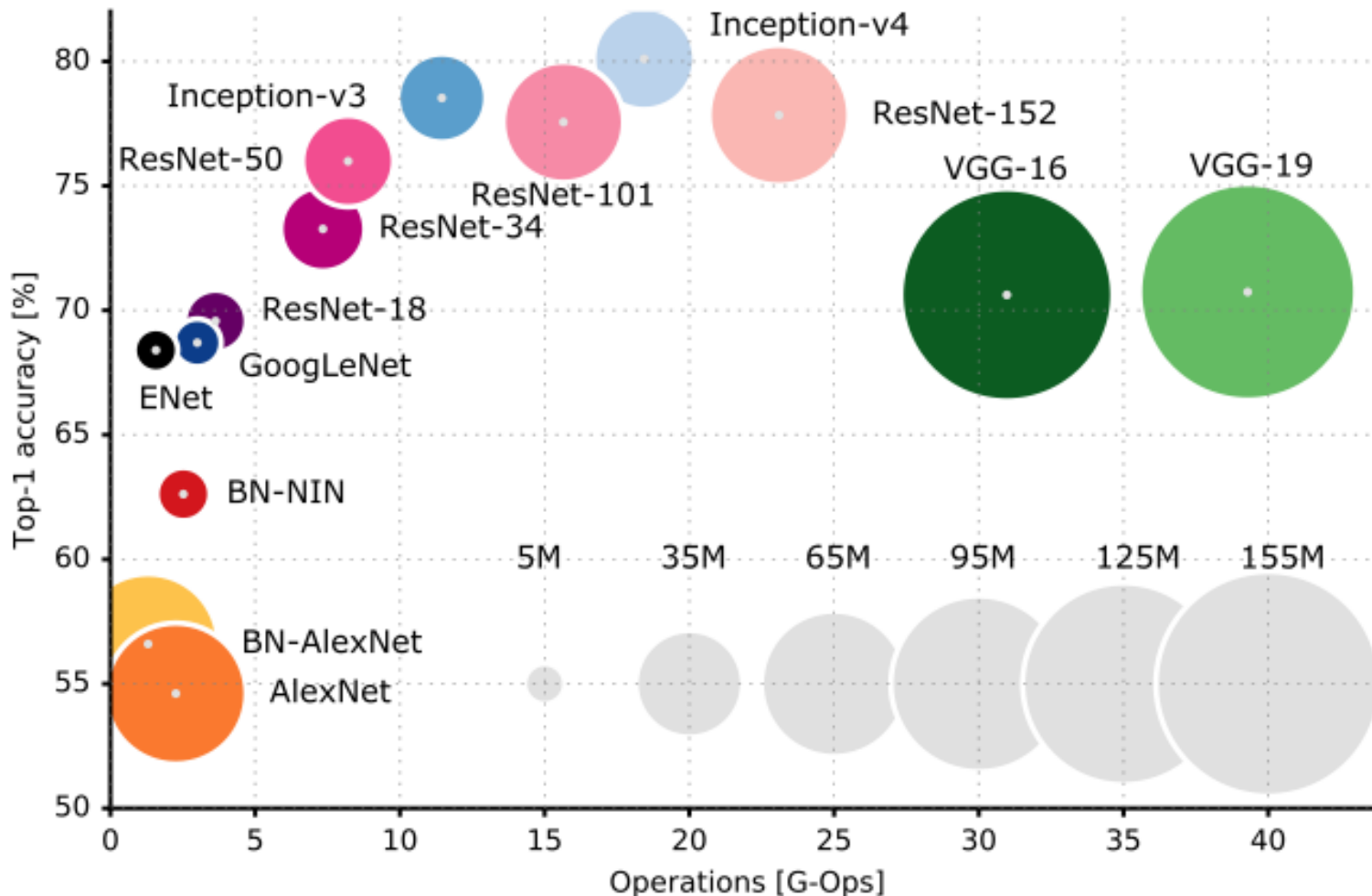
# Convolutional Neural Networks (CNNs)

- An illustration of the most popular deep learning architectures is provided in http://josephpcohen.com/w/visualizing-cnn-architectures-side-by-side-with-mxnet/



LeNet  AlexNet  VGG  GoogLeNet  Inception  Resnet

# CNNs: Example

- How to create (and instantiate) one CNN (Sequential):

```python
def cnn_model(input_shape=(32, 32, 3)):

    model = Sequential()

    model.add(Conv2D(filters=32, kernel_size=3, padding='same', activation='relu', input_shape=input_shape))
    model.add(Conv2D(filters=32, kernel_size=3, padding='same', activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Conv2D(filters=64, kernel_size=3, padding='same', activation='relu'))
    model.add(Conv2D(filters=64, kernel_size=3, padding='same', activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Conv2D(filters=64, kernel_size=3, padding='same', activation='relu'))
    model.add(Conv2D(filters=64, kernel_size=3, padding='same', activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Flatten())
    model.add(Dense(512, activation='relu'))
    model.add(Dense(10, activation='softmax'))


    return model

# ##########################################################
# Instantiate model
model = cnn_model()
model.summary()

# Compile model
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
```

("**Sequential**" objects provide the simplest way. "**Functional**" objects enable additional functionalities)

CNNs: Example

- How to use (or fine tune) one well known CNN model:
  - Example: Inception.V3

```python
def create_inception(tot_classes):
    imgs_input = Input((args['image_height'], args['image_width'], 3))

    if args['fine_tuning'] == 0:
        model_tmp = inception_v3.InceptionV3(input_shape=(args['image_height'], args['image_width'], 3),
                            weights=None, include_top=False)
    else:
        model_tmp = inception_v3.InceptionV3(input_shape=(args['image_height'], args['image_width'], 3),
                            weights='imagenet', include_top=False)
        model_tmp.trainable = False

    x = model_tmp(imgs_input, training=False)

    x = keras.layers.GlobalAveragePooling2D()(x)
    outs = Dense(tot_classes, activation='linear')(x)

    md = Model(inputs=imgs_input, outputs=outs)
    md.compile(optimizer=RMSprop(learning_rate=args['learning_rate']), loss=tf.keras.losses.MeanAbsoluteError())
    return md
```

- This is typically the approach that attains the best results.
  - Not only the architecture was coherently designed, but also the weights were optimized based in huge datasets.

CNNs: Example

- How to train one CNN:

```python
# For small datasets
history = model.fit(X_train, y_train, batch_size=32, epochs=10, verbose=1, validation_split=.3)

# For large datasets
for i in range(tot_batches):
        [X_batch, y_batch] = get_input_batch(i)
        loss = model.train_on_batch(X_batch, y_batch)
```

- Typical preprocessing steps:

```python
# Images are typically normalized to the range [0, 1].
X_train = X_train.astype("float32") / 255
X_test  = X_test.astype("float32") / 255


# In classification problems, labels are typically converted to one-hot encoding.
y_train = to_categorical(y_train)
y_test  = to_categorical(y_test)
```